CS 331 Midterm Exam 2

Friday, April 29, 2016

Please bubble your answers in on the provided answer sheet. Also be sure to write and bubble in your student ID number (without the leading 'A') on the answer sheet.

1. What is the time complexity for locating an element with a given value in a doubly-linked list of *N* elements sorted in ascending order?

   (a) $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

   (d) $O(N \log N)$

2. What is the time complexity for popping off the first (i.e., oldest) element pushed onto a stack which contains *N* elements?

   (a) $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

   (d) $O(N \log N)$

3. One way to reverse the sequence of elements in a queue would be to dequeue them and push them onto a stack one by one, then pop all the elements off the stack while enqueueing them again. What would be the time complexity of performing this sequence of operations with a queue of *N* elements?

   (a) $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

   (d) $O(N \log N)$

4. What is the time complexity of inserting a single element into a heap of *N* elements and ensuring that heap ordering semantics are maintained?

   (a) $O(1)$

   (b) $O(\log N)$

   (c) $O(N)$

   (d) $O(N \log N)$

5. What is the time complexity of removing all elements in descending order from a heap containing *N* elements?
   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N \log N)$
6. What is the time complexity for removing the root element from an unbalanced binary search tree containing N elements?
   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N \log N)$
7. What is the time complexity for removing the root element from an AVL tree containing N elements?
   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N \log N)$
8. What is the time complexity of fixing a “RL” imbalance in an AVL tree resulting from the addition of a new node? (Ignore the time taken to actually locate the insertion spot for the node.)
   (a) $O(1)$
   (b) $O(\log N)$
   (c) $O(N)$
   (d) $O(N \log N)$
9. Which of the following applications is a heap NOT particularly well suited to?
   (a) implementing heapsort
   (b) implementing a priority queue
   (c) permitting rapid access to the largest of a partially ordered set of elements
   (d) maintaining a full ordering of elements across multiple insertions and deletions
10. Which of the following data structures is best suited for use in simulating a “fair”, first-in-first-out scheduling system?
    (a) a stack
    (b) a queue
    (c) a priority queue
    (d) an AVL tree

11. Consider the non-balanced binary search tree constructed from the following (ordered) sequence of values:

```
9 10 8 1 3 2
```

What is the value stored in the root of the tree?

(a) 1

(b) 2

(c) 9

(d) 10

12. What is the height of the tree from problem (11)?

(a) 5

(b) 4

(c) 3

(d) 2

13. If inserting the value 7 into the tree constructed in problem (11), where would it go?

(a) Above node (1)

(b) As the right child of node (3)

(c) As the right child of node (2)

(d) As the left child of node (10)

14. If deleting the value 8 from the tree constructed in problem (11), how should we best go about updating the tree?

(a) Set the left child of (9) to None

(b) Set the left child of (10) to None

(c) Set the left child of node (9) to node (1)

(d) Set the right child of node (1) to node (9), and make (1) the root of the tree

15. Consider the following (ordered) sequence of values used to construct a binary search tree:

```
2 4 3
```

What sequence of rotations (if any) would be required to balanced the resulting tree?

(a) a left rotation about (4)

(b) a left rotation about (2)

(c) a right rotation about (4) followed by a left rotation about (2)

(d) no rotations are needed

16. Consider the following (ordered) sequence of values used to construct a binary search tree:

```
5 4 6
```

What sequence of rotations (if any) would be required to balanced the resulting tree?

(a) a left rotation about (6)

(b) a right rotation about (4)

(c) a left rotation about (4) followed by a right rotation about (5)

(d) no rotations are needed

17. Consider the *balanced* AVL tree constructed from the following (ordered) sequence of values:

```
2 4 8 5 7 9
```

How many rotations were needed over the course of adding all the values to keep the tree balanced?

(a) 2

(b) 3

(c) 4

(d) 6

18. What types of rotations were performed while constructing the tree in problem (17)?

(a) 2 right rotations

(b) 3 left rotations

(c) 3 left rotations, 1 right rotation

(d) 2 left rotations, 4 right rotations

19. Which of the following values, if added to the tree constructed in problem (17), would require additional rebalancing (through one or more rotations)?

(a) 0

(b) 3

(c) 6

(d) 10

20. Which of the following best translates a potentially negative index `idx` into a positive offset into a list data structure (containing `self.count` elements), per Python's regular array index semantics?

(a)
```
idx = (abs(idx) + self.count) % self.count)
```

(b)
```
idx = idx + (-self.count)
```

(c)
```
if idx < 0:
    idx = idx + self.count
```

(d)
```
if idx < 0:
    idx = self.count - idx
```

21. Which choice completes the following implementation of `__len__` for a circular linked list, where `self.head` refers to the sentinel head?

```
def __len__(self):
    count = 0

    ____________________________________
    return count
```

(a)
```
n = self.head
while n:
    count += 1
    n = n.next
```

(b)
```
n = self.head.next
while n is not self.head:
    count += 1
    n = self.head.next
```

(c)
```
n = self.head.prior
while n is not self.head:
    count += 1
    n = n.next
```

(d)
```
n = self.head.next
while n is not self.head:
    count += 1
    n = n.next
```

22. Which of the following implements a generator-based iterator for a circular linked list, where `self.head` refers to the sentinel head?

(a)
```
yield self.head.next.val
```

(b)
```
n = self.head.next
while n:
    yield n.next.val
```

(c)
```
yield from self.head
while iter(self):
    yield n.val
    n = next(self)
```

(d)
```
n = self.head.next
while n is not self.head:
    yield n.val
    n = n.next
```

23. Given that `self.head` refers to the sentinel head link of a circular, doubly-linked list implementation, which choice completes the following function so that it removes just the *last* occurrence of x from the list?

```
def remove_last(self, x):

    ____________________________________
```

(a)
```
n = self.head.prior
while n.val != x:
    n.next = n.next
    n.prior = n.prior
    n = n.prior
```

(b)
```
n = self.head.next
while n is not self.head:
    if n.val == x:
        n.prior.next = n.next
        n.next.prior = n.prior
        return
    n = n.next
```

(c)
```
n = self.head.prior
while n is not self.head:
    if n.val == x:
        n.prior.next = n.next
        n.next.prior = n.prior
        return
    n = n.prior
```

(d)
```
n = self.head.next
while n.val != x:
    n = n.next
while n is not self.head:
    if n.val == x:
        break
    n = n.next
n.prior.next = n.next
n.next.prior = n.prior
```

24. Which choice completes the following function so that it returns `True` only for strings that contain properly balanced pairs of parentheses, brackets, and curly braces?

```
def check_parens(str):
    pairs = {'(': ')', '[': ']', '{': '}'}
    stack = Stack()
    for c in str:
        if c in pairs.keys():
            stack.push(c)
        elif c in pairs.values():

            _____________________________________
    if stack:
        return False
    else:
        return True
```

(a)
```
if c == pairs[stack.pop()]:
    return True
```

(b)
```
if not stack:
    return False
```

(c)
```
while stack.pop() != c
    if not stack:
        return False
```

(d)
```
if not stack or c != pairs[stack.pop()]:
    return False
```

25. In a circular, array-backed queue implementation, which of the following implements the “dequeue” operation? (Assume the queue is not empty)

(a)
```
val = self.data[self.head]
self.head -= 1
return val
```

(b)
```
val = self.data[self.head]
self.head = (self.head + 1) % len(self.data)
return val
```

(c)
```
val = self.data[self.head]
self.head = (self.head - 1) % len(self.data)
return val
```

(d)
```
val = self.data[self.head]
self.head = self.tail - self.head + 1
return val
```

26. Which of the following correctly removes the maximum value from a heap and re-establishes heap ordering semantics? (Assume that the `_heapify` method works correctly as described in class, and is passed a starting index.)

(a)
```
def del_max(self):
    for i in range(len(self.data)-1):
        self.data[i] = self.data[i+1]
    self._heapify(0)
```

(b)
```
def del_max(self):
    self._heapify(len(self.data)-1)
    for i in range(len(self.data)):
        self.data[i] = self.data[i+1]
    del self.data[len(self.data)-1]
```

(c)
```
def del_max(self):
    self.data[0] = self.data[len(self.data)-1]
    del self.data[len(self.data)-1]
    self._heapify(0)
```

(d)
```
def del_max(self):
    self.data[0] = self.data[len(self.data)-1]
    del self.data[len(self.data)-1]
    self._heapify(len(self.data)-1)
```

27. Which of the following implements returns the total number of elements in a binary tree rooted at the node passed in as the initial argument `t`?

(a)
```
def tree_size(t):
    if not t:
        return 0
    else:
        return 1 + tree_size(t.left) + tree_size(t.right)
```

(b)
```
def tree_size(t):
    if not t:
        return count
    else:
        return tree_size(t.left) + tree_size(t.right)
```

(c)
```
def tree_size(t, count=0):
    if not t:
        return count
    else:
        return tree_size(t.left, count+1) + tree_size(t.right, count+1)
```

(d)
```
def tree_size(t, count=0):
    if not t:
        return count
    else:
        return tree_size(t.left, count) + tree_size(t.right, count+1)
```

28. Which of the following conditions returns true for a “LR” imbalance at node n in a binary search tree?

(a)
```
(BSTree.height(n.left) > BSTree.height(n.right)+1 and
 BSTree.height(n.left.left) > BSTree.height(n.left.right))
```

(b)
```
(BSTree.height(n.left)+1 < BSTree.height(n.right) and
 BSTree.height(n.right.left) < BSTree.height(n.right.right))
```

(c)
```
(BSTree.height(n.left) > BSTree.height(n.right)+1 and
 BSTree.height(n.right.left) < BSTree.height(n.left.right))
```

(d)
```
(BSTree.height(n.left) > BSTree.height(n.right)+1 and
 BSTree.height(n.left.left) < BSTree.height(n.left.right))
```

29. Which of the following returns the *successor* of `x` from the binary search tree rooted at `t`, where the successor of a value is defined as the smallest value larger than (but not equal to) that value? If no successor exists, `None` is returned; the initial value of t is assumed to not be `None`.

(a)
```
def successor_rec(t):
    if x >= t.val and t.left:
        return successor_rec(t.left)
    elif x < t.val and t.right:
        s = successor_rec(t.right)
        if t.val > s:
            return t.val
        else:
            return s
    elif x < t.val:
        return t.val
    else:
        return None
```

(b)
```
def successor_rec(t):
    if x >= t.val and t.right:
        return successor_rec(t.right)
    elif x < t.val and t.left:
        s = successor_rec(t.left)
        if s:
            return s
        else:
            return t.val
    elif x < t.val:
        return t.val
    else:
        return None
```

(c)
```
def successor_rec(t):
    if x > t.val:
        return successor_rec(t.right)
    elif x < t.val and not t.left:
        return t.val
    else:
        return successor_rec(t.left)
```

(d)
```
def successor_rec(t):
    if not t:
        return None
    return min(successor_rec(t.right),
               successor_rec(t.left))
```

30. Which of the following methods fixes a RR-imbalance in a tree about the node passed in as `t`?

(a)
```
def fix_rr(t):
    l = self.left
    t.val, l.val = l.val, t.val
    t.left, l.left, t.right, l.right = l.left, l.right, l, t.right
```

(b)
```
def fix_rr(t):
    r = t.right
    t.val, r.val = r.val, t.val
    t.left, r.left, t.right, r.right = r, t.left, r.right, r.left
```

(c)
```
def fix_rr(t):
    l  = self.left
    ll = self.left.left
    self.left = ll
    l.right, ll.left = ll.left, l

    r = self.right
    t.val, r.val = r.val, t.val
    t.left, t.right, r.left, r.right = r.left, r, r.right, t.right
```

(d)
```
def fix_rr(t):
    r  = t.right
    rr = t.right.right
    t.right = rr
    r.right, rr.left = rr.left, r

    r = t.right
    t.val, r.val = r.val, t.val
    t.left, r.left, t.right, r.right = r, t.left, r.right, r.left
```